APPENDIX C

UNIX™ System V Init and Getty

UNIX™ System V Init and Getty

1. Introduction

In the UNIX* system environment, the initial process spawning is controlled and overseen by the first process forked by the UNIX operating system as it comes up at boot time. This process is known as *init*. One of the major jobs of *init* is to fork processes which will become the *getty-login-sh* sequence. This sequence of processes allows users to *login* and takes care of setting up the initial conditions on the outgoing terminal lines so that the speed and the other terminal related states are correct. *Init* and these other processes also keep an accounting file **/etc/wtmp** that is available to processes on the system. With these files it is possible to determine the state of each process that *init* has spawned, and if it is a terminal line, who the current user is. One program in particular, *who*(1), provides a means of examining these files.

This document describes the capabilities of each program used in this new implementation, the databases involved, and how to create and maintain these databases. In addition, the debugging features designed into both *init* and *getty* are described in the event remedial action is required or modifications are attempted.

2. Init

Init is driven by a database, its previous internal level, its current internal level, and events which cause it to wake up.

2.1 The Database: /etc/inittab

Init's database, kept in the file **/etc/inittab**, consists of any number of separate entries, each with the form:

id:level:type:process

id — The *id* is a one to four letter identifier which is used by *init* internally to label entries in its process table. It is also placed in the dynamic record file, **/etc/utmp**, and the history file, **/etc/wtmp**. The *id* should be unique.

level — The *level* specifies at which levels *init* should be concerned with this entry. *Level* is a string of characters consisting of [0-6a-c]. Anytime that *init*'s internal level matches a level specified by *level*, this entry is *active*. If *init*'s internal level does not match any of the levels specified, then *init* makes certain that the process is not running. If the level field is empty it is equivalent to the string "0123456".

type — The *type* specifies some further condition required for or by the execution of an entry.

- off — The entry is not to run even if the levels match.
- once — The entry is to be run only if *init* is entering a level. This means if *init* has been awakened by powerfail or because a child died this entry will not be activated. Only when a user signal requests a change of *init*'s internal state to a state which is different from its current state, and this new state is one in which this entry should be active, will this entry be activated.

---

* UNIX is a Trademark of Bell Telephone Laboratories, Incorporated.

wait — *Wait* has all the characteristics of *once*, plus it causes *init* to wait until the process spawned dies before reading anymore entries from its database. This allows for initialization actions to be performed and completed before allowing other processes which might be affected to start running. It is common in the OSS environment for shared memory segments to be initialized this way and semaphores to be conditioned.

respawn — *Respawn* requests that this entry continue to run as long as *init* is running in a level which is in this entry's *level* field. Most processes spawned by *init* fall into this category. All *getty* processes are marked as *respawn*. Whenever *init* detects the death of a process that was marked *respawn*, it spawns a new process to take its place.

boot — *Boot* entries have the execution behavior of *once* entries. They are started only when *init* is switching to a numeric run state for the first time. Most commonly *boot* entries have an empty *level* string, meaning that no matter which level *init* switches to the first time, the *boot* entry will be run. Should there be a more specific *level* string, for example "01", then the *boot* entry would only be run if *init* switched to either the 0 or 1 run state as its first numeric level.

bootwait — *Bootwait* entries have the execution behavior of *wait* entries and they, like *boot* entries, are only run as *init* switches to a numeric level for the first time.

power — *Power* entries act like *once* entries and are activated if *init* receives a SIGPWR signal (19) and is in a state which matches the active states for the entry.

powerwait — *Powerwait* entries act like *wait* entries and are activated if *init* receives a SIGPWR signal and is in a state which matches the active states for the entry.

initdefault — *Initdefault* is a non-standard entry in that it does not specify some process to be spawned. Instead it only specifies which level *init* is to go to initially when it is coming up at boot time. This allows the system to be rebooted without an operator having to make entries at the system console if so desired. If there is no *initdefault* entry, then *init* will ask at the system console, **/dev/syscon**, for the initial run state. In addition to specifying the numbered states, the single-user state [s] may also be specified.

process — The *process* field is the action that *init* will ask a *sh* to perform whenever the entry is activated. The string in the **process** field is given a prefix of "exec " so that each entry will only generate one process initially. *Init* then forks and execs

```
sh -c "exec process"
```

This means that the **process** string can take full advantage of all **sh** syntax. The only peculiarities arise from the string "exec ", which was prefixed to the string, and because initially there is no standard input, output, or error output. The addition of "exec " to the string means that if the user wants to have a single entry generate more than one process, for example making a list of the people on the system at the time of a powerfail and mailing it to **root** by the command "who | mail root", it would have to be put in as

```
pf::powerwait:sh -c "who | mail root"
```

to work. If it was put in simply as "who | mail root", it would be executed as "exec who | mail root", and only the *who* process would be created before the *sh* disappeared. The

lack of standard input and output channels must be addressed by explicitly specifying them. An example is the **blog** program that many OSS's run as a *bootwait* entry as the system comes up. Since it requires the operator to supply input, it appears as

bl::bootwait:/etc/blog </dev/syscon >/dev/syscon 2>&1

in **/etc/inittab**.

2.2 Levels

A level is one of seven numeric levels, denoted 0, 1, 2, 3, 4, 5, or 6, three temporary levels, denoted a, b, or c, or the single-user level, s. Normally *init* runs in a numeric level. Precisely how a particular level is used depends entirely on the database and the system administrator. The temporary levels allow certain entries to be started on demand without affecting any processes that were started at a particular level. The temporary levels immediately revert to the previous numeric level once all entries in the database have been scanned to see if they should be started at the temporary level. When an entry is started by a switch to a temporary level, it becomes independent of future level changes by *init*, except a change to the single-user level. The only way to kill a process that was started as a respawnable demand process, without going to the single-user level, is to modify the database, declaring the entry to be *off*.

The single-user level is the one level independent of the database. For this reason it is not a level in the normal sense. In the single-user level *init* spawns off a *su* process on the system console, and that is the only process that it maintains while at the single-user level. The single-user level can be entered at two different places in *init*. If it is entered at boot time it allows the operator to look over the file systems without having *init* attempt to do any file I/O, which might cause further problems. *Init* will not attempt to recreate **/etc/utmp** or access **/etc/wtmp** until after it has left this initial single-user level. If the single-user level is entered at any other time, *init* does do the bookkeepping in the record files.

The system administrator requests *init* to change levels by running a secondary copy of *init* itself. **/etc/init** is linked to **/bin/telinit**, and it is usually through the *telinit* name that this is accomplished. *Init* can only be run by root or a privileged group. Whenever *init* starts running and finds that its process id is not 1, it assumes that it is a user initiated copy, which is supposed to send a signal to the real *init*. The usage is:

telinit [0123456sSqQabc]

and the single character argument specifies the signal to be sent to *init*. If the request is to switch to the single-user level, 'S' or 's', then *init* also relinks **/dev/syscon** to the terminal originating the request so that it becomes the virtual system console, thus insuring that future messages from *init* will be directed to the terminal where the operator is located. When it does this relinking it also sends a message to **/dev/systty**, saying that the console is being relinked to some other terminal so that there is a record of the fact at the physical system console.

2.3 Waking Events

There are four events which will wake *init*: boot, a powerfail, death of a child process, or a user signal.

boot — *Init* operates in the **boot** state until it has entered a numeric state for the first time. It is not possible for *init* to reenter the boot state a second time. Commands labeled *boot* and *bootwait* are executed when changing to a numeric state for the first time, if the levels match.

powerfail — Any time power fails, the operating system sends a SIGPWR signal to all processes. *Init* will execute commands with types of *power* and *powerfail*.

child death — Any time a child process of *init* dies, *init* receives a SIGCLD signal (18). The dead child process may be one of two types, a direct decendent of *init*, or a process whose own parent process died before it did. The parent of a process automatically

becomes *init*, if its real parent should die before it does. *Init* determines immediately if the defunct process was one of its own children or an *orphan*. If it was one of its own, it performs the necessary bookkeepping on its internal process table to note that the process died. If *init* was busy at the time it received the SIGCLD signal, it then returns to complete whatever action it was performing. If *init* was asleep, it then scans its database to determine if any other actions should be taken, such as respawning the process.

user signal *Init* catches all signals that it is possible for a process to catch. Most signals have specific meaning to *init*, usually requesting it to change its current state in some way. There is one signal, the 'Q' signal, which is used just to waken *init* and cause it to scan its database. This is often issued after a change has been made to the database so that *init* will put the new change into effect immediately. If this was not done, the change would not become effective until *init* had wakened for some other reason. Other than during the initialization phase, it is solely with signals that the system administrator controls the internal level at which *init* is running.

2.4 Normal Operational Behavior

Init scans **/etc/inittab** once or twice for each event which wakes it up. If it is in the *boot* or *powerfail* state, it scans the table once, looking for entries of these types, and then switches itself back to a *normal* state and scans again.

Its first action in the normal state is to scan **/etc/inittab** and remove all processes which are currently active and should not be at the current level. *Init* employs one of two methods when killing its child processes depending on whether it is changing levels or not. If *init* is not changing levels, it forks a child process for each child that needs to be killed, and has that child process send the signals to the process targeted for extinction. Killing a process involves sending it two signals. First a SIGTERM signal (15), is sent so that it can clean up after itself and die gracefully. After waiting the amount of time defined as TWARN (the default value is 20 seconds), a SIGKILL signal (9), is sent, which guarantees that the child will die, if it hasn't done so already. Forking a child to do the killing has the advantage that the main *init* process need not wait for all the processes it is killing to die before beginning the spawning of new processes. The disadvantage is that if many processes were being killed this way, there would be a very real chance of the operating system process table filling up, which causes the *fork* system call to fail. This in turn would upset *init* at the very least and cause it to have to wait anyway. For this reason, when *init* is changing levels, it assumes that it may have many processes to terminate and so it sends the signals itself, waits for the required 20 seconds, and sends the final termination signals, before continuing. Once the old processes have been removed, *init* makes an entry in its accounting files if it is changing levels. At this point it either enters the single-user level or rescans its database looking for processes that need to be spawned at the current level and in the current state. In the normal state of operation *init* is looking for entries whose types are *off*, *once*, *wait*, or *respawn*.

With the completion of the scan of the database in the normal state, *init* is ready to wait for another event. To ensure that a user who just logged off has had his or her files updated to the disk and to insure that the bookkeepping is also updated to the disk, *init* performs a *sync* system call and then pauses until it is awakened again for some new reason.

If *init* finds that it is being requested to switch to the single-user level when it wakens from the pause, it saves all the *ioctl* information about the system console in the file */etc/ioctl.syscon* before proceeding to remove all its other children. It does this so that if the system is being taken down, the new *init* process will know how to set up the system console to talk to it. It is a convenient feature to not have to change the baud rate and terminal specifications if you are rebooting a system remotely. Because *init* preserves the ioctl state of the system console across system reboots, messages coming out during reboots are legible to the operator, no matter where the system console happens to be linked.

All written messages from *init* are sent to */dev/syscon*. In reality, *init* itself does not send the message, but forks a child to send the message. This is because *init* must never open a terminal line or it will be assigned a controlling terminal. Since *init* has no controlling terminal, it can spawn *getty* processes which initially have no controlling terminal. When such a *getty* opens its assigned terminal, the terminal becomes the controlling terminal for it and its children. In the one instance *init* needs input from the system administrator during the initialization phase. In this case, the child process which is asking for the run level opens */dev/systty*, which is always the physical system console, before opening */dev/syscon*, the virtual system console. This causes */dev/systty* to be the child's controlling terminal. Thus, should the computer be coming up, */dev/syscon* not be linked to */dev/systty*, and */dev/syscon* be down (perhaps because the datalink went down during the reboot), it is possible for a person at */dev/systty* to regain control by typing a <DEL> character. This causes a SIGINT signal (2) to be sent to the child process, which will relink */dev/systty* to */dev/syscon* and ask again for a run level, this time at the physical system console.

2.5 Setting Tunable Variables

Init has several tunable timing constants that can be adjusted when it is compiled.

SLEEPTIME *Init* guarantees that it will awaken occasionally even if the system is quite inactive. It does this by setting an alarm timer before going to sleep. The length of that timer is defined by SLEEPTIME, and is initially five minutes. Since *init* does a *sync* system call each time it wakes, this guarantees that there will be a *sync* at least once every SLEEPTIME seconds.

TWARN TWARN is the number of seconds between the SIGTERM signal and the SIGKILL signal, when *init* is removing processes. It should be set long enough so that all processes who want to, can die gracefully on receipt of the SIGTERM signal. It is initially 20 seconds.

NPROC This is the size of the internal process table *init* uses to keep track of its child processes. It currently defaults to 100, though it can be passed in during compilation with the -D option. I recommend you set it to the size of the system's process table.

WARNFREQUENCY To prevent *init* from flooding the system console with error messages when it own internal process table is full, *init* only generates an error message once each WARNFREQUENCY times that it is unable to find a slot. Proper sizing of the internal process table should prevent this condition from ever occurring.

Init cannot directly tell if there is something wrong when it tries to fork and exec a command. It assumes that there is something wrong if it has to respawn a particular entry too often. There are three related defines controlling this feature, SPAWN_LIMIT, SPAWN_INTERVAL, and INHIBIT.

SPAWN_LIMIT SPAWN_LIMIT is the number of times a process may respawn in a certain interval of time before further respawns are inhibited.

SPAWN_INTERVAL SPAWN_INTERVAL is the interval of time in seconds that SPAWN_LIMIT number of respawns must occur to cause inhibition of an entry. If an entry should respawn too often, a message is generated on the system console indicating which line in **/etc/inittab** is at fault.

INHIBIT INHIBIT is the number of seconds of inhibition that will be applied to a process which has respawned too often.

SPAWN_LIMIT and SPAWN_INTERVAL should be set so that it is possible for *init* to respawn a process fast enough to cause inhibition, but not so low that it is possible to have a legal death of a process happen so rapidly that it is inhibited. The current limits are ten respawns in two minutes. The real problem is that when something like *getty* disappears, *init* becomes active trying to respawn many processes and never gets to respawn a single process often enough to set off the alarm. The INHIBIT limit is five minutes. Once an entry is inhibited, it is possible to restart it sooner than

INHIBIT seconds later by sending *init* the 'Q' signal. The normal problem is a typo in **/etc/inittab**, and the normal procedure is to correct the typo and then do a "telinit Q" to cause *init* to attempt the spawning entry again.

2.6 Debugging Features

Init has some debugging features built in. There are three conditional debug flags, which allow various flavors of debugging to be enabled.

UDEBUG This flag causes *init* to be compiled in a form that can be run as a normal user process instead of as process 1. This allows a person to use *sdb* on it in a normal fashion and to not disturb the rest of the system while debugging or modifications are made and tested. There are differences in this user version of *init*. It assumes that **utmp**, **wtmp**, **inittab**, **ioctl.syscon**, and **debug** are all in the local directory instead of /etc. It also writes to **/dev/sysconx** and **/dev/systtyx**, instead of **/dev/syscon** and **/dev/systty**. It does not process all signals in the same fashion that the real *init* does. Signals SIGINT, SIGQUIT, SIGIOT, and SIGTERM, which correspond to the signals to change to levels 2, 3, 4, and ignore are left in their default modes, so that it is possible to terminate the user "init" from a terminal. Signals SIGUSR1 and SIGUSR2, which are normally ignored by the real *init* are set to cause an *abort* for capturing cores of the debug *init*. The UDEBUG flag automatically sets the DEBUG flag, meaning that the first level of debug will be generated by the *init* and written into the file *debug* in the current directory.

DEBUG This flag causes a version of *init* to be produced that can be run as the real *init*, but which generates diagnostic messages about process removal, level changes, and accounting and writes them in the file **/etc/debug**.

DEBUG1 DEBUG1 causes the diagnostic output generated by DEBUG1 to be increased substantially. Specifically it produces messages about each process being spawned from *inittab*.

3. Getty

Getty is responsible for making appropriate setting of terminal characteristics and baud rate so that a user can communicate with the UNIX system. The most important of those features is the choice of a baud rate so that input and output make sense. In the old version of *getty*, there was a hardwired table in *getty* which controlled the search for the correct speed. The starting point in the search is specified by the arguments passed to *getty*. If there was some reason to change the baud rate search, *getty* had to be modified itself, and recompiled. In the new *getty*, the search is controlled by an ascii file, **/etc/gettydefs**, and changing or augmenting the search behavior only requires that the file be edited.

3.1 Usage

Getty is normally started from **/etc/inittab** by *init*. *Getty* takes from one to six arguments:

getty [-h] [-t time] line [speed_label] [term_type] [line_disc]

−h This switch tells *getty* that it should not drop the Data Terminal Ready signal before resetting the line. This switch currently only works in the CB-UNIX system environment. Normally *getty* ensures that DTR goes down so that connections to the Develcon dataswitch will be disconnected everytime. The EIA protocol requires that a dataset see DTR drop and be reasserted before answering another call. It is possible for *getty* to come back on a line before all the processes spun off by the previous user have died and closed their connections to the line. In this case, DTR would not drop if *getty* didn't insure it. This switch is required for programs like *ct*, which initiate a call from the computer to a user (instead of the user calling the computer), putting a *getty* on the resulting connected line. Without the -h switch, the *getty* would immediately disconnect the user again.

−t — This switch specifies that the *getty* should die after the specified number of seconds if nothing is typed. This prevents datasets from being tied up if someone isn't actually logging in after they've gotten connected.

line — *Line* is the name of the terminal line, which *getty* is to open and set up. It is minus **/dev/** since *getty* does a *chdir* to the **/dev** directory and expects to find it in that directory.

speed_label — The *speed_label* is usually something like "1200" or "9600", which appears to directly specify a baud rate, but in reality can be anything since it really is a label of an entry in **/etc/gettydefs** for which *getty* looks. It specifies the entry *getty* will start with when trying to find an appropriate speed to for the terminal. It defaults to "300" if there is none given.

term_type — The *term_type* specifies which terminal discipline is to be used. If this is specified, the virtual terminal protocol becomes immediately effective on the line. Typical types might be "vt100", "hp45", or "tek". Whatever type is specified, it must be a terminal handler that has been compiled into the operating system to be effective. This argument is given for lines that are hardwired to the computer.

line_disc — The *line discipline* is the last thing that can be specified. The most common is "half" or "half_duplex", when there is a half duplex terminal coming into the computer. This causes the appropriate line discipline to be associated with the line.

3.2 The Database: /etc/gettydefs

Whenever *getty* is invoked it references its database to determine certain information about how to set up the line. Each entry in the database has a fixed format.

label# initial flags # final flags # login msg #nextlabel

Getty matches its *speed_label* argument against the "label" field. It stops searching when it finds an entry with a label that matches. The entry specifies how the terminal is supposed to be setup during the initial phase, the phase when *getty* prints out the "login msg" and reads in the user's login name, and the final phase, when *getty* exec's the *login* program to continue to the *login* process. The baud rate is specified as an *ioctl* flag in both the initial and final flags fields.

The flags themselves are strings matching the define variables found in **/usr/include/termio.h**. It should be noted that these flags may be partially or totally overridden if there is a terminal type specified. When a terminal type is enabled, it resets various flags to suitable conditions automatically.

During the initial phase, *getty* always puts the terminal into a non-echoing *raw* mode. This allows it to take each character as it comes in and infer certain things about the terminal. For instance, if it sees upper case alphabetic characters, but no lower case, it then assumes that the terminal is upper case only and sets it up in the final configuration so that the upper to lower case conversions are made. Also if the speed is wrong it will get a <NULL> character (or <ESC><NULL> character if a terminal type is set) if there is a framing or parity error. This means that the speed is wrong and another speed should be tried.

The typical "initial flags" would only include the speed, for example "B1200 CS7 PARENB HUPCL". "CS7 PARENB" sets the line for 7 bits, even parity characters. "HUPCL sets the line to hangup on close. Typical final flags would be "B1200 SANE IXANY TAB3". "SANE" is not a real flag found in the header file, but a collection of *ioctl* flags used for normal terminal behavior. "IXANY" permits the use of any character to restart output. "TAB3" says to expand tabs on output.

The "login msg" field is the message that *getty* will print before waiting for the user to enter his or her login name. It may contain anything desired and *getty* understands normal special character conventions so that "\n" means <lf> as does "\012". On systems that are not using the terminal

handlers and where lines are hardwired, people have been known to make up special entries for different terminal types, for example:

```
vt100-2400# B2400 # B2400 SANE TAB3 7953OGIN: #vt100-1200
        # 33[H 33[2JAMACCS System B
```

where the "login msg" contains the special vt100 characters required to clear the screen. Notice also that the entry can take more than one line. Entries are delimited by a blank line. Lines that begin with a pound sign (#) are ignored so that comments may be added to the file.

The "next_label" field tells *getty* which entry to try next if it gets an indication that the speed is wrong. In the above example it would look for an entry with the name "vt100-1200" if this one wasn't at the proper speed. Normally the entries don't contain terminal specific information, and the various speed choices are linked together in a closed circle of some sort. For example it is common to have 9600 -> 4800 -> 2400 -> 1200 -> 300 -> 9600. In this way, no matter where you enter the circle, sooner or later you should be able to get to the speed that is correct for your terminal.

To enable the system administrator to check the database for readability by *getty*, there is a checking mode in which *getty* can be run.

```
getty -c gettydefs_like_file
```

When *getty* is run in this mode, it scans the entire input file specified and deciphers each entry, printing out the resulting modes that it will set. If it finds a line that it cannot read, it prints an appropriate message, which allows the administrator to correct the entry. By this mechanism it is possible to avoid installing a misformatted gettydefs file and have it tie up the system.

Also as a safety measure, should *getty* be unable to find **/etc/gettydefs**, it does have a one fallback entry built in. Should *gettydefs* disappear for some reason, a user could still log in at 300 baud, since this is the default setting in the built-in entry.

3.3 Operational Behavior

As has been shown earlier, *getty* sets up a line as specified by an entry from **/etc/gettydefs** and from any additional arguments, outputs the "login msg" field, and then tries to read the user's login name from the line. During the input of the login name, getty checks for speed mismatches that the operating system will report as a <NULL> character. If such a mismatch occurs, *getty* tries the next speed specified by the current entry, and repeats the whole sequence. Also while reading in the login name, *getty* makes a guess whether the terminal is upper case only. If it sees some upper case characters, but no lower case characters, it assumes that the terminal is upper case only and sets the *ioctl* state of the line to translate upper case letters to lower case on input, and lower to upper case on output.

An addition has been made to *getty* and *login*, which allows for environmental variables to be set up at the time a user enters his or her login name. This allows users to control the behavior of their *.profile* at the time they specify their login names. *Getty* executes the *login* program by passing all the separate words given it in response to the login message as arguments to *login*. If for example, the user responded with "jls f", then *getty* would execute "login jls f" as its final action. See the *login* section to see how this modifies the commands behavior.

4. login

Unlike *init* and *getty*, *login* did not require a great deal of modification. The only required change was that it should write to **/etc/utmp** and **/etc/wtmp** in the new format. This change was minor. At the time this change was made, a change visible to the user was also made: the ability to add to the environment. This change was added as a convenience. It allows the user to modify the behavior of his or her *.profile* by having environmental variables set which the *.profile* script knows about.

The basic change was that any additional words provided in response to the basic "login:" query are placed in the environment of the *sh* executed by *login* as its last act in the following way. If the word does not contain an '=', a shell variable of the type "Ln=word" is created. "n" is a number starting at 0 and for each new environment variable it is incremented by one. If the word does contain an '=', then the whole string is passed in the environment unchanged. For example, "TERM=2621" would be placed in the environment unchanged and the shell variable $TERM would be defined as "2621".

To preserve security, there are a couple of exceptions. It is not possible to change the shell variables $PATH or $SHELL by this mechanism. That means that a restricted shell will remain restricted and that the user cannot gain access to commands that might allow him to avoid the usual restrictions of *rsh*.

5. who

Who(1) is the program that reads the history files maintained by *init*, *getty*, and *login*. Since the format of these files was changed substantially, it was necessary to change *who*. In the process some additional features where added to *who* so that it would convey more useful information to users. The standard usage for *who* is:

who [-uTlpdbrtas] [[am i] or [utmp_like_file]]

u This returns a listing of useful information for all the users. This information includes login time, activity, pid and comment from inittab file.

T Report the writability state of the terminal for that entry.

l Report all entries that are living *getty* processes.

p Report all entries for living children of *init* excluding *getty* and decendents of *getty*.

d Report all the entries for processes that have died.

b Report the boot time entries that *init* has made. In **/etc/utmp** there is only one such entry.

r Report the run level entries that *init* has made. In **/etc/utmp** there is only one such entry, the current run level entry. The current state, the number of times in that state, and the previous state are also reported.

t Report the change of date entries that have been made by the *date*(1) command when the clock was reset. These are required in the history file, **/etc/wtmp**, if accounting is to be done.

a Report all the entries.

s Report information for all users in short form, this is the default.

If no file is specified, then **/etc/utmp** is assumed. The **who am i** sequence returns the entry for the user typing the command.

There are various output formats for the different kinds of entry. In particular, entries for users and *getty* processes list the amount of time since output to the terminal occurred. This is often of interest since it shows other users whether someone is actually working at a terminal or not. The comment field at the end of the entry from **/etc/inittab** is also included, which can conveniently be set up to be the location of the terminal. Dead entries report the exit status for the process that died. This can be of use, since it shows whether the process terminated abnormally or not.

6. Other Affected Programs

All programs accessing the accounting files were affected by the new utmp structure. In particular, *date*(1) makes two entries indicating the old time and new time, whenever it changes the system clock. Also affected are the commands in **/usr/lib/acct**, which produces reports based on the information in **/etc/wtmp.**

7. utmp format

A major change in going to the new *init* was that it uses a different format in writing out its records in **/etc/utmp** and **/etc/wtmp.** The new format is:

```
/*      <sys/types.h> must be included.                                 */

#define UTMP_FILE     "/etc/utmp"
#define WTMP_FILE     "/etc/wtmp"
#define ut_name       ut_user

struct utmp
 {
        char ut_user[8] ;       /* User login name */
        char ut_id[4] ;         /* /etc/lines id(usually line #) */
        char ut_line[12] ;              /* device name (console, lnxx) */
        short ut_pid ;                  /* process id */
        short ut_type ;         /* type of entry */
        struct exit_status
          {
            short e_termination ; /* Process termination status */
            short e_exit ;      /* Process exit status */
          }
        ut_exit ;               /* The exit status of a process
                                        * marked as DEAD_PROCESS.
                                        */
        time_t ut_time ;        /* time entry was made */
 } ;

/*      Definitions for ut_type                                         */

#define EMPTY           0
#define RUN_LVL         1
#define BOOT_TIME       2
#define OLD_TIME        3
#define NEW_TIME        4
#define INIT_PROCESS    5       /* Process spawned by "init" */
#define LOGIN_PROCESS   6       /* A "getty" process waiting for login */
#define USER_PROCESS7           /* A user process */
#define DEAD_PROCESS    8
#define ACCOUNTING      9

#define UTMAXTYPE   ACCOUNTING  /* Largest legal value of ut_type */

/*      Special strings or formats used in the "ut_line" field when  */
/*      accounting for something other than a process.               */
/*      No string for the ut_line field can be more than 11 chars +  */
/*      a NULL in length.                                            */

#define RUNLVL_MSG  "run-level %c"
#define BOOT_MSG    "system boot"
#define OTIME_MSG   "old time"
#define NTIME_MSG   "new time"
```

The *ut_type* field completely identifies the type of entry, the *ut_id* field only contains the "id" as found in the "id" field of **/etc/inittab.** The *ut_line* field was expanded and freed so that it can

contain things like *console* or other things that are not of the form **/dev/ln**xx. Finally *ut_exit* contains the exit status of processes that *init* has spawned and that have subsequently died.